DPTree: A Distributed Pattern Tree Index for Partial-Match Queries in Peer-to-Peer Networks

Dyce Jing Zhao, Dik Lun Lee, and Qiong Luo

Department of Computer Science,
Hong Kong University of Science & Technology, Hong Kong
{zhaojing, dlee, luo}@cs.ust.hk

Abstract. Partial-match queries return data items that contain a subset of the query keywords and order the results based on the statistical properties of the matched keywords. They are essential for information retrieval on large document repositories. However, most current peer-to-peer networks for information retrieval are based on distributed hashing and as such cannot support partial-match queries efficiently. In this paper, we describe an efficient and scalable technique to support partial-match queries on peer-to-peer networks. We observe that the combinations of keywords in the queries are only a small subset of all possible combinations of the keywords in the documents. Therefore, we propose a distributed index structure, called a distributed pattern tree (DPTree), to record frequent query patterns, i.e., combinations of keywords, learnt from the query history at each node in the network. Using this index, a query can identify its best matching patterns quickly and data lookup can be done in logarithmic time with respect to the network size. Our simulation studies on the TREC data sets have shown promising results in comparison with other previous approaches.

1 Introduction

While the decentralized nature of peer-to-peer file sharing systems enables robustness and scalability, it also poses great challenges for resource lookup in these systems. Most existing peer-to-peer approaches do not support complex queries efficiently. Unstructured peer-to-peer systems maintain no forward knowledge for remote computers. As a result they are essentially in the dilemma between network coverage and bandwidth cost while searching, whether using simple or complex queries. Structured peer-to-peer systems, such as Chord [6] and Pastry [2], are mostly based on distributed hash tables (DHT). They determine the hosting peer(s) of a data item by applying hash functions on the descriptors (e.g., the keys) of the data item. Therefore, they can quickly route a query to the destination where matching data items can be found, but they only allow exact match, i.e., the query and the descriptors of the data items must be identical. Exact match does not meet the needs of full-text keyword queries. It is difficult for such structured peer-to-peer systems to support more complex queries such as partial-match queries efficiently.

Y. Ioannidis et al. (Eds.): EDBT 2006, LNCS 3896, pp. 515–532, 2006.

Generally speaking, a partial-match query returns data items that contain the query keywords and ranks the results according to the matched keywords. For example, a full-text retrieval system accepts keywords as the query, and retrieves and ranks documents containing the keywords. There are many ranking functions. In this paper, we adopt the inner product similarity because it ranks documents purely based on the matched keywords without considering other non-matching keywords in the documents. Inner product is not only simple but also suitable for users who are looking for some relevant documents (i.e., most web search users). For example, assessors for TREC relevance judgments consider a document to be relevant to a query if any slice of the document is relevant to the query [21].

We note that most existing peer-to-peer networks for information retrieval are based on distributed hashing and as such cannot support partial-match queries efficiently. In this paper, we propose to develop a distributed index called *DP-Tree* which supports full-text partial-match queries efficiently on peer-to-peer networks. The idea is that each node manages a list of relevant documents for popular queries, and organizes the document lists to be searchable within $O(\log N)$ time where N is the total number of participating nodes. In this paper, we use the term *pattern* to represent the (unordered) set of keywords that a query contains. While the number of possible patterns is astronomical, given the large and ever-growing document repositories nowadays, we observe that only a small portion of the patterns are frequently used in the queries. This observation motivates us to focus on frequent patterns mined from the query history. In fact, query history has been utilized successfully in many peer-to-peer search systems to improve the performance [8, 11, 22].

To support the organization of patterns and pattern mining, we developed the distributed pattern trees (DPTree). By definition, a DPTree is a tree structure that can be implemented on one or more computers: a node can be implemented on more than one computer, or alternatively, the whole tree can be implemented on one computer. Each DPTree node corresponds to a pattern. In particular, the root of a DPTree represents a single-word pattern, its children are responsible for 2-word patterns, and its grand children correspond to 3-word patterns, etc. Each node maintains an index to the list of documents matching the pattern that the node maintains. For clarity, we hereafter use the terms *DPTree node* to refer to the node itself, the *pattern* it maintains or the machine (or machines) that implement the node when no ambiguity arise.

A DPTree node is capable of initiating, forwarding and responding to queries. During the search procedure, a DPTree node selectively records a query history, from which frequent patterns can be mined periodically. A DPTree starts with a single-word pattern (i.e., the root node) and is expanded and adapted dynamically based on the frequent patterns found. The roots of the DPTree's form an addressable network using distributed hash tables. By applying mining technique on query history, our approach is able to answer most queries quickly and precisely by managing a suitable number of frequent patterns. In addition, we employ *random access sequence* on patterns to establish strict mapping between

a pattern and the DPTree that it resides. This eliminates redundant patterns across DPTree's without breaking the storage and network load balance among the peers. Another data structure called *sub-tree summary* enables a DPTree node to estimate its entire sub-tree in an economical way, which spares overlay maintenance cost.

We conducted simulation over TREC data and compared our system with other two systems [3, 17] which are to be introduced in the next section. The experimental results show that our approach achieves significant gain on search effectiveness and efficiency.

The remainder of this paper is organized as follows: Section 2 introduces some related works. Section 3 describes the basic DPTree approach in detail. We discuss some improvements for the basic approach in section 4 concerning redundancy and maintenance. Section 5 presents our experimental results and Section 6 summarizes our work.

2 Related Work

To our current knowledge, no authoritative peer-to-peer approaches were found for partial-match full-text queries. However, concerning the larger domain of similarity search, there are some impressive works [3, 4, 13].

pSearch [3][4] and SSW [13] use Latent Semantic Indexing (LSI) to map documents into a semantic vector space and perform search based on the Euclidian distance between the query point and the document points. In especial, pSearch is developed on top of CAN. In addition to the use of LSI, pSearch applies rolling index and register a document to p places in the CAN using p separate partial semantic spaces. This reduces the dimensionality and therefore enables CAN to manage full-text documents. In SSW, computers form clusters, each of which manages non-overlapping regions of the semantic vector space. A cluster is split into two at a certain cluster size when new nodes join the network. Every computer in a cluster knows its region and splitting history, which are used to compute a unique ID for the cluster. All the clusters form a circle with clockwise ascending cluster ID's. A query message computes a partial cluster ID using available splitting history and hops along the circle in a greedy manner until it reaches the cluster with the complete ID. Query routing is efficient in both pSearch and SSW. pSearch and SSW split successively the vector space into cells and position data points according to the cells that they reside in. These approaches work well with similarity metrics such as cosine or Euclidian distance. They are, however, inherently not applicable to partial match. This is because although various document ranking metrics (e.g., inner product) can be applied to process partial-match queries, none of the metrics follow the triangle rule (i.e., $d_{AC} \leq d_{AB} + d_{BC}$ for any three points A, B,and C, where d_{XY} is the distance between point X and point Y.). This indicates that documents relevant to a query are not guaranteed to be similar to each other. As a result, the basic assumption of pSearch and SSW does not hold that data points relevant to a query reside in a small number of adjacent cells.

Efforts were made to address the particular issue of partial-match query [5, 10, 15, 17, 18]. One approach [15] assigns every keyword set that appears in the network a computer which indexes a list of relevant documents for the keyword set. A document is said to be relevant to a keyword set if all of its keywords co-exist in at least one slice (or a window) of the document. While this method does partial-match search quickly, it bears large storage and maintenance overhead since, with no selectivity, it will possibly supervise a huge number of keyword sets.

Another approach [17] applies joins in distributed database to work with partial-match search. It maintains a list of documents for each single keyword. To compute the result set for a multi-keyword query consisting of more than one keywords, it starts with the first keyword and locates quickly a list of relevant documents. A bloom filter is computed based on the document list retrieved which is much smaller in size compared to the list of relevant documents. The bloom filter is sent to the next keyword along with the query. Upon receiving the query and the bloom filter, the computer responsible for the next keyword will integrate the bloom filter and its document indices into a new relevant list. This method is efficient for small data sets. However, it is shown [9] that the bloom filter consumes significant bandwidth cost in large-scale networks.

3 Partial-Match Search Using Distributed Pattern Trees

Our observation is that compared to the huge number of possible keyword sets, only a very small portion of them are frequently used as queries. Therefore, it is unnecessary as well as infeasible to create a document list for every possible pattern. This motivates us to use distributed pattern trees that extend themselves from query history. The DPTree's manage a tree hierarchy of popular query patterns. Every node for a DPTree is associated with a pattern and is represented by a cluster of strongly connected computers responsible for a pattern. The parent-child relationship between two DPTree nodes indicates the containment relationship between their patterns. A pattern P_2 is said to contain another pattern P_1 if and only if all keywords that appear in P_1 also exist in P_2. The root of a DPTree is a pattern of a single keyword. The root nodes are positioned using distributed hash table while the single-word patterns that they manage serve as the key. Among a few applicable DHT's [1, 2, 6], we choose Chord [6] for placement of pattern tree roots.

During a series of query sessions, every DPTree node selectively collects its query history and mines the frequent patterns periodically. The DPTree's, initially consisting of only roots, are then expanded dynamically as new frequent patterns are discovered. A keyword based search starts from one single word clusters and is propagated along the pattern trees until the patterns that best match the query are reached.

3.1 Overlay Formation

We now discuss the construction of the overlay network. In essence, our approach uses distributed pattern trees on top of the Chord protocol. Figure 1 displays a

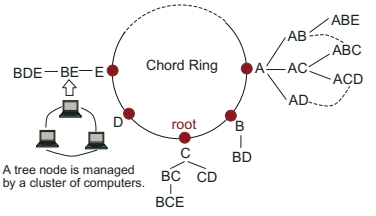


Fig. 1. Distributed pattern trees positioned along a Chord ring

simple example of the network containing five DPTree's, rooted at A, B, C, D, and E, respectively. Each node is labelled with the pattern that it manages.

In the initial state, a DPTree contains only the root, which corresponds to a single-word pattern. The DPTree's are generated dynamically. When a DPTree node N_i receives a query, it checks if it fully matches the query. If so, it answers the query. If not, it checks if any of its child nodes matches the query. If a match is found, the query is forwarded to the corresponding child node for processing. Otherwise, N_i is considered the best matching node and the answers are returned from N_i. Since N_i does not fully match the query, it records the query in its query history. Algorithm 1. describes this query logging procedure.

Algorithm 1. Query logging during a search session

Input:
N is a node of a DPTree
Q is a query message
P is the peer that initiates a query

Procedure: $query(N, Q, P)$
1: **if** N fully matches Q **then**
2: $retrieve(N, Q, P)$
3: **else**
4: **if** $\exists N'$, N' is a child of N **and** N' matches Q **then**
5: $query(N', Q, P)$;
6: **else**
7: $retrieve(N, Q, P)$;
8: log Q in the local query history

Procedure: $retrieve(N, Q, P)$
9: flood Q within the cluster of computers for N;
10: return the highest-ranked documents to P;

A DPTree node, say node N_0, monitors its own query history. It mines periodically the frequent query patterns using any pattern mining methods such as Apriori [19] and Eclat [14]. If N_0 contains t words, the pattern mining process mines all frequent patterns containing $t + 1$. A new node is created for each of the mined frequent patterns and becomes the child of N_0.

In the DPTree, the parent and child nodes can communicate with each other and know each other's patterns. Suppose N_0 manages pattern P_0. To create a child node, N_1, whose pattern is P_1 ($P_0 \subset P_1$), the list of documents maintained by N_0 is split such that documents that match N_1 are moved to N_1. In addition, queries in N_0's query history that are longer than P_0 are moved N_1.

Recall that DPTree nodes are distributed on a set of machines and that a DPTree node can be implemented on more than one machine. To minimize the network cost for DPTree node splitting and to balance the overlay maintenance overhead among the peers, we use a splitting strategy as follows,

1. for every machine involved, count the number of its indexed documents that match P_1 and rank the machines using their counts;
2. assign machines to the new node N_1 in descending order of the machines' ranks;
3. stop splitting when the storage for the document indices are roughly balanced between the machines managing node N_0 and those managing node N_1.

Node N_0 notifies the creation of the new node N_1 to its parent. The parent is responsible for two tasks:

1. it checks if the same pattern exists by polling all of its children with pattern P_1; if the same pattern is found, it asks the two corresponding tree nodes to merge into one;
2. if P_1 does not exist in the children, the parent continues to look for children that match a sub-pattern of P_1 and build unidirectional links from the matching children to N_1.

The unidirectional links (shown as dotted curves in Figure 1) are used to ensure that every pattern of the pattern tree can be reached from the root in a greedy manner. These links lower the search cost by relaxing the strict tree structure of DPTree.

At the initial stage, there may be insufficient query history available for pattern tree generation. As an alternative, a pattern tree can extend itself by mining the frequent patterns based on the label of its indexed documents.

3.2 Maintenance and Search

A machine or a peer in our peer-to-peer network is capable of initiating two classes of operations: the maintenance operation when a peer joins or leaves the network and the search operation when a peer submits a query. In this section, we discuss how these operations are performed in the order that they appear during the lifetime of a peer.

The maintenance and search operations involve four types of messages. Table 1 lists the information that the four types of messages carry, where *foreign index* means the index for a document on a remote peer.

Among these messages, peer join and document registration messages are for the peer joining operation; query and peer leave messages are for search and peer leaving operations, respectively.

Table 1. Description of messages used in the system

Message Name	Message Information
peer join	peer label, peer address
document registration	document label, registration key, peer address
query	keyword list, peer address
peer leave	foreign indices, query history, peer address

Peer Joining. When a peer P_{new} joins a network, it first computes the label for itself and generates a peer join message. While a peer label can be defined in various ways, we use the centroid of the peer's local document set as the peer's label. This method takes advantage of data locality and query locality, and hence helps to reduce maintenance and search costs.

P_{new} sends its peer join message to an existing peer P_0 randomly selected in the network. The message is then directed to a DPTree root N_r which matches the most frequent word in P_{new}'s local repository. N_r can be located by following the tree edges from P_0 to P_0's tree root, N_{r0}, and along the Chord ring from N_{r0} to N_r. This operation is efficient because:

1. the height of a DPTree is small, since DPTree height is bounded by the lengths of the user queries which are typically short [23];
2. searching on the Chord ring takes only logarithmic time with respect to the number of pattern trees, and only the tree roots are positioned along the Chord ring.

Upon receiving P_{new}'s join message, N_r computes its *recruiting priority* with respect to P_{new}. The recruiting priority of all of N_r's child nodes are also computed. The recruiting priority between a tree node N and P_{new} indicates how likely P_{new} is going to join N. Formally, N's recruiting priority is defined as follows.

$$rp(P_{new}, N) = L_N * Sim(P_{new}, N), \qquad L_N = AR_N * (FI_N / Cap_N),$$

where $Sim(P_{new}, N)$ denotes the similarity between P_{new} and C. The label of a DPTree node is the pattern that it manages. The factor L_N in the equation evaluates N's workload for maintaining foreign indices and for processing query requests. To compute L_N we use N's recent access rate AR_N (i.e., the number of messages received/forwarded/returned during a time unit) and the *consumption ratio* (i.e., the percentage of storage that's already consumed) of N's local storage which is represented by N's current storage for foreign indices, FI_N, over N's capacity, Cap_N. This indicates that the possibility of a new peer joining a DPTree node is subject to two factors: the peer's relevance to the tree node and the maintenance overhead of the node. Therefore, the use of recruiting priority helps balance the load among the tree nodes.

If N_r gets the highest recruiting priority, P_{new} joins the cluster responsible for N_r. Otherwise, the peer join message is forwarded to the child of N_r which has the highest recruiting priority and the join process continues in a similar way. After the peer joins a tree node, it prepares the document registration messages for each of its local documents. The registration key in a document registration message is a word appearing in the document and is used to find a pattern tree root. P_{new}'s documents are then published to all of the relevant tree nodes using the document registration messages.

Search. The search consists of two steps. In the first step, when a peer P initiates a query, the query message is propagated to the DPTree roots that match one of the query keywords. A query is routed from the starting peer to the relevant roots in the same way as a peer join message is routed. Note that it is possible that a query word does not match any of the DPTree's on the Chord ring. In this case, a failure message is returned since the query word is obviously beyond the global vocabulary.

In the second step, a separate search process is executed on each of the relevant DPTree's. Upon receiving a query message, a DPTree node (or, more precisely, a peer responsible for the DPTree node) first checks whether it is the most similar node to the query, and responds to the query if it is. Otherwise, the query is propagated to a randomly selected child node that is more similar to the query. The detailed procedure is described in Algorithm 1. in Section 3.1.

When the second step completes, we are able to identify the tree nodes that best satisfy the query, although they may not be perfect matches. When a tree node decides to answer a query, it uses the document labels of its foreign indices to compute the relevance score and returns the top M results, where M is a pre-defined number. If multiple DPTree's are contacted during a search process, the query initiator will do a local re-ranking after all query results are returned.

Peer Leaving. When a peer leaves the network, it hands its foreign indices and query history to one of its neighbors in the cluster. If a leaving peer is the last peer in the cluster, it contacts a neighboring cluster which bears the lowest maintenance overhead and asks the neighbor to take over its task.

In addition to the activities mentioned previously, a peer may fail unexpectedly. Our system is insensitive to single peer failure due to the use of clusters. As an alternative, the foreign indices can be replicated within a cluster. Should a peer fail, the peers within the same cluster will seamlessly take over its job.

4 Improvements

After we describe the basic model, we are now able to estimate theoretically the performance of our DPTree approach. The network costs can be formulated as follows.

$$CSearch = O(\log N' + QLength + 2 * M); \tag{1}$$

$$CJoin = O(\log N' + (\log N' + TSize) * D * W). \tag{2}$$

We use N' to denote the number of clusters that manage DPTree roots, and use N to denote the number of peers in the overlay network. N'/N is about $1/3$ in our experiments. Equation (1) summarizes the search cost for a query with $QLength$ query words if M query results are to be returned. Equation (2) shows the join cost for a new peer with D documents and on average W words per document. $TSize$ is the average number of nodes contained in a DPTree. The first item in the equation is the overlay maintenance cost, while the second item is the document registration cost.

The above equations for cost estimation suggest DPTree's superiority in searching. However, they also reveal the non-trivial cost for peer joining or leaving. In this section, we propose two methods, *random access sequence* and *sub-tree summary*, to cope with the considerable maintenance cost.

4.1 Random Access Sequence

The DPTree-based network contains a certain degree of redundancy. For example, a query with three words A, B and D may be directed to DPTree(A) as well as DPTree(B). In a network as shown in Figure 1, both node AD under DPTree(A) and node BD under DPTree(B) will have query ABD in their query history. Therefore, it is possible that the pattern ABD will appear in more than one DPTree, as shown in blue in Figure 2.

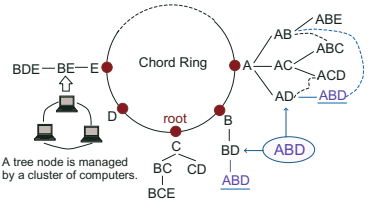


Fig. 2. Redundancy in the network

To eliminate redundant patterns, we generate a random access sequence (RAS) for every multi-keyword query. RAS determines the order that the query keywords are processed. When a query is initiated, the search process generates the random access sequence of the query, and contact DPTree's according to the order of the single keywords listed in RAS. Similarly, when a new frequent pattern is mined, RAS is used to determine which DPTree is going to create a node for the new pattern. Given a multi-keyword query Q, the random access sequence is generated as described in Algorithm 2.

The use of random access sequence avoids redundant patterns across DPTree's. As a result, the cost for document registration and search is decreased. In addition, applying RAS does not break the balance of the storage and network workload because the access sequence of a query is random. Therefore, although the search process favors some DPTree's over others with respect to

Algorithm 2. Random access sequence

Input:
Q is a query message
Output:
Q whose keywords are re-ordered
Procedure: $RAS(Q)$
1: order the keywords in Q alphabetically;
2: $srandom(Q)$
3: s = number of keywords in Q;
4: **for** i=s to 2 **do**
5: $p = [random()\ mod\ (i-1)] + 1$;
6: swap the i th keyword and the $(i-p)$th keyword in Q;
7: **return** Q;

a single query, globally the storage and network overheads of the DPTree's are not biased.

4.2 Sub-tree Summary

It is shown that the bulk of the peer joining cost comes from document registration. The reason is that in the document registration procedure, when a document tries to find relevant patterns in a DPTree, it has to explore every node of the DPTree, which involves a large amount of data transmission among the peers. We use a data structure called *sub-tree summary* to avoid unnecessary DPTree node access and as such to reduce the document registration cost by avoiding unnecessary DPTree node access.

In essential, every DPTree node keeps a summary of the sub-tree under it. The summary of a sub-tree contains the following information:

- the location of the sub-tree root;
- for every node N in the sub-tree, a $< P_N, R_N >$ pair, where P_N is the pattern for node N and R_N is the minimum relevance score that N permits.

A document is forwarded to node N if the relevance score between the document and P_N is greater than R_N. Using the sub-tree summaries, a new document will have to access only the DPTree nodes along the path to a relevant node.

To estimate the relevance score threshold R_N for a pattern tree node N, the following information is used:

- The relevance score of every foreign index that N contains.
- The maximum number of foreign indices that N will maintain, denoted as M'_N. N is set to be proportional to the number of queries that N does not perfectly match but answered.
- An amplification factor $\gamma(\gamma > 1)$ which is universal for all nodes.

We assume that with respect to a pattern P_N, the relevance score distribution of all its relevant documents follows the Zipf's law. We use the current set of

foreign index to approximate the distribution curve. We set the maximum foreign index number with two values: M'_N and $M_N = \gamma * M'_N$. We use M'_N and M_N to compute two relevance score thresholds, R'_N and R_N, respectively ($R'_N > R_N$). R_N is sent to N's parent node for sub-tree summarizing. We will explain later in this section why we use two thresholds.

The sub-tree summaries are built as follows:

- **DPTree node creation.** When a DPTree node N_0 creates a new child N, it creates a summary for the resulting sub-tree containing only N. The summary contains the location of node N and $< P_N, 0 >$. When the sub-tree summary for any DPTree is updated, the node propagates the update to its parent, whose sub-tree summary is updated consequently.
- **Foreign index update.** A node N knows its relevance score threshold R_N which is used for sub-tree summarizing.
  - When a foreign index is deleted from node N, N uses the resulting new list of foreign indices to compute the new values for its two relevance score thresholds R'_{N_new} and R_{N_new} ($R'_{N\_new} > R_{N\_new}, R_N > R_{N\_new}$). If $R'_{N\_new} < R_N$, we update the pair $< P_N, R_N >$ with $< P_N, R_{N\_new} >$ for all nodes along the path from N to the pattern tree root.
  - When a foreign index is inserted into node N, nothing is done. However, when a document reaches a node N but is rejected, N will compute the updated value for its relevance score thresholds R_{N_new} and replace $< P_N, R_N >$ with $< P_N, R_{N\_new} >$ if $R_{N\_new} > R_N$.

This method greatly reduces the number of peers accessed for document registration, and thus reducing the registration cost. However, it introduces overheads for sub-tree summary update. We minimize the sub-tree summary update cost by setting a looser relevance score threshold than the actual estimation. This allows the sub-tree summaries to be updated only after a number of successive index deletions have been done. It should be noted that the value of the amplifying factor γ is a trade off between the redundant peer access cost and sub-tree summary update cost.

5 Experiments

To evaluate the proposed distributed index, we compare our method to the Bloom filter approach [17] which is specifically for partial-match queries and pSearch [3] which is a well-studied peer-to-peer search method.

We apply the vector space model and label a document as a term vector. "Term frequency inverse document frequency" (TFIDF) is used to compute the weight of a term in a document. Since it is impractical to obtain the global document frequency in a dynamic peer-to-peer system, we use the local document frequency instead. Both our DPTree and the bloom filter method are built on top of the Chord protocol.

Given a keyword based query, the goal of search is to find a specified number of documents that are most relevant to the query.

5.1 Simulation Setup

The document set consists of about 500,000 documents taken from Volumes 4[1] and 5[2]of the TREC collection, consisting of about 500,000 documents. The query keywords were generated from the global keyword database according to their document frequencies in the Web repository maintained by the UC Berkeley and Stanford Dig- ital Library projects (See http://elib.cs.berkeley.edu/docfreq/), which consists of 49,602,191 pages.

All programs were written in Java (JDK 1.2.0) and run on a PC with 2.5G Pentium 4 processor and 512M memory.

We used the following metrics in the simulation:

1. **Effectiveness** is measured by the average precision and recall. We define the *hit list* for a query as a list of all available documents on the network that match the query. Let the hit list be H and the returned result list be R for any query. Precision is defined as $\frac{|R \cap H|}{|R|}$, and recall is defined as $\frac{|R \cap H|}{|H|}$.
2. **Search Path Length** is defined as the average number of logical hops traversed by a query message before it reaches the destination.
3. **Search cost** is defined as the average number of messages that a query incurs in the search process.
4. **Maintenance cost** is defined as the average number of messages used to handle peer activities including peer joining and leaving.
5. **Storage cost** is defined as the average number of foreign indices that a peer maintains.

The simulation parameters, their range of values and default settings are specified in Table 2.

Table 2. Simulation parameters

	Description	Range	Default
N	Number of peers in the network	1k - 20k	10k
n	Number of document per peer	1 - 20	5
L	Length of queries	1 - 5	
M	Number of document returned		20
λ	Number of operations per round		100
w	Number of warm-up queries used	0 - 5k	1k

We used a large number of peers in the simulation to evaluate the scalability of the three methods. The number of keywords in a query ranges from 1 to 5 with a

[1] TREC Volume 4, May 1996 Collection includes material from the Financial Times Limited (1991, 1992, 1993, 1994), the Congressional Record of the 103rd Congress (1993), and the Federal Register (1994).

[2] TREC Volume 5, April 1997 Collection includes material from the Foreign Broadcast Information Service (1996) and the Los Angeles Times (1989, 1990).

uniform distribution. This was to approximate the real-word query lengths [16]. We set the number of returned documents to 20, which is the typical number of documents that most web users are willing to examine. To generate a new peer, we varied the number of documents per peer from 1 to 20 and assigned documents randomly selected from the TREC collection. As a result, duplicated documents may exist in our simulation. Since our method applies mining techniques in building up the distributed index, a longer warm-up period would likely yield better search performance. To examine the effect of the mining techniques, experiments with no warm-up queries and with a rich query history were run.

5.2 Comparison

We conducted extensive experiments to compare our work with two other approaches: the Bloom filter approach and pSearch. According to the configuration in [3], we let pSearch take 4 partial semantic spaces, and the dimension of each partial semantic space was $2.3log_N$ where N was the network size. Considering the unavailability of a global document set, we randomly picked a subset of 5,000 documents from the TREC collection and LSI via singular value decomposition (SVD) was applied to to the subset to generate the semantic space. For simplicity, we denote our DPTree method by *DPTree* and the Bloom filter approach by *BLF* (BLF for bloom filter) in the later experiments.

First, we compared the search effectiveness of the three methods. Figure 3 presents the precision-recall curve for networks with 1,000 to 20,000 nodes. Our method (DPTree) yields a much higher search quality especially when the number of peers in the network is large. Its retrieval precision is about 35% better than both of the other methods when the network size $N = 20k$. This was due to the use of the distributed pattern trees. By mining the frequent patterns dynamically, our approach adapts user needs and is insensitive to network size. It should be noted, however, that inner-product was used as the similarity measure

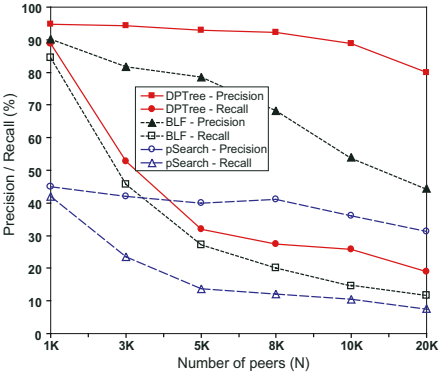


Fig. 3. Precision-recall curve

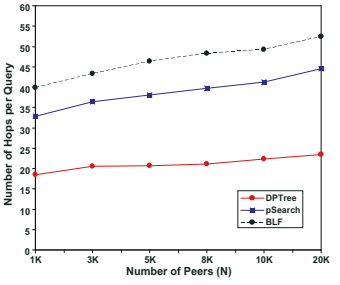


Fig. 4. Search Path Length

in the experiments and that since pSearch was not designed for partial-match, it is expected to yield poor performance.

In Figures 4, 5, and 6 we examined the scalability of the three methods in terms of the search cost, search path length and system maintenance cost, respectively.

Search path length is an important performance measure since it affects the query response time. shows that the search path in our method is shortest. To identify the destination for a query, our system takes almost only half of the number of hops compared to the other two approaches. The search path length was logarithmic with respect to the network size for all of the three methods, but our method carried a smaller constant term since DHT was applied to the tree roots instead of the entire set of peer computers in our system.

Figure 5 depicts the search cost in terms of the total number of messages transferred for a query. Our methods outperformed the other two especially when sufficient query history was available. With our default setting (N = 10k,

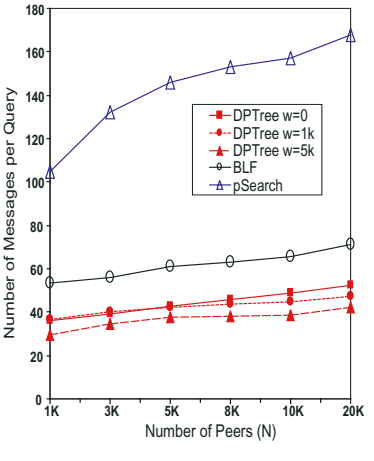


Fig. 5. Search cost

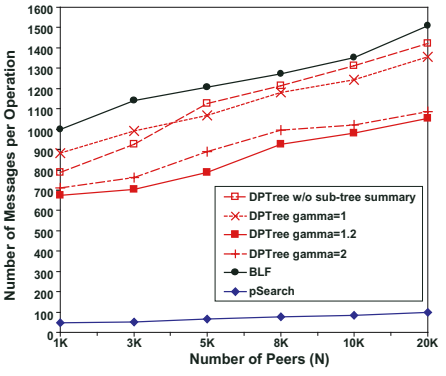


Fig. 6. Maintenance for various number of peers

w = 1k), DPTree on average spent about 70% less messages than pSearch and about12% less than BLF. pSearch incurred a much higher cost during search since it performed four separate searches over the entire network for every query.

We evaluate the average maintenance cost of each method by setting the rates that peers joined and left the network to be the same and varying the network size from 1k to 20k nodes. No query was performed during this round of experiments so that the system maintenance cost was isolated from the search cost. Figure 6 presents the maintenance cost of the three methods. The effect of sub-tree summary for DPTree was also measured, with the amplification factor γ set to 1, 1.2 and 2. It can be observed that when the γ is at 1.2, applying sub-tree summary can reduce the maintenance cost by 25% compared to our basic approach (see Figure 6). Thus the effectiveness of sub-tree summary is justified. However, Figure 6 also shows that except for pSearch, both our approach and the bloom filter approach incur a considerable maintenance overhead when peer membership changes very frequently.

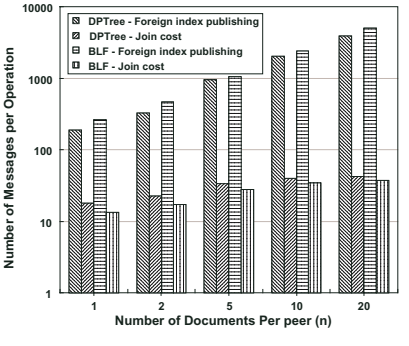
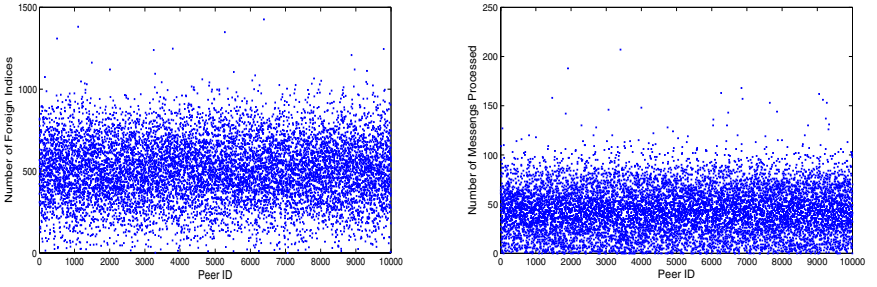


Fig. 7. Maintenance for various number of documents per peer



(a) Distribution of storage (b) Distribution of network load

Fig. 8. Distribution of storage and network load among peers

To further analyze the network maintenance cost, we varied the number of documents that a peer held from 1 to 20, and displayed in Figure 7 the overlay maintenance cost and the document registration cost separately. The y-axis of Figure 7 is displayed in logarithmic scale. The experimental result indicates that the vast majority of the maintenance cost for peer membership changes comes from foreign document registration. Therefore, although our method incurs non-trivial network cost in a highly dynamic environment, we can apply various techniques to reduce the cost. For example, lazy update can greatly reduce document registration cost when some peers keep joining and leaving the network frequently. Moreover, document registration can even be suspended during a period of heavy network traffic since it does not affect the correctness of the overlay. As a result, we believe that our approach could scale in terms of query efficiency, search cost as well as maintenance cost.

To estimate the workload distribution among all the peers, we plotted the network and storage loads with respect to the peer ID's in Figure 8. Figure 8(a) displays the number of foreign indices that each peer maintains, and it shows that the storage load for most peers was close to the average load. In addition, the maximum number of foreign indices per peer does not exceed 1,500 while the average number is 514. Figure 8(b) displays the number of messages processed during a certain time period for 10,000 peers. It shows that the distribution of network load is balanced among peers.

6 Conclusion

In this paper we proposed a distributed index that supports partial-match search. We developed the distributed pattern trees that record query history in a selective way and extend themselves by mining frequent patterns from the query history. The roots of the pattern trees are positioned on the overlay network using any distributed hash table method. We proposed the random access sequence and sub-tree summary techniques to decrease the maintenance cost.

Experiments showed that our approach yields high precision for keyword-based partial-match queries. Our method was also proven to be efficient in query

routing. The performance of data look-up improves after a certain warm-up period. It was also shown that our approach achieves good load balance. Although peer membership changes incur a considerable maintenance overhead, the majority of the costs comes from foreign-index publishing, and the network cost for the peer join operation was small. We argue that foreign-index publishing can be suspended in a heavy traffic period and that peer join and data lookup are not affected by this suspended operation. Moreover, applying lazy update can further reduce the maintenance cost. As a result, our approach is scalable.

References

1. B.Y. Zhao and J.D. Kubiatowicz, A.D. Joseph, *Tapestry: An Infrastructure for Fault-tolerant Wide-area Location and Routing*, Technical Report UCB/CSD-01-1141, U. C. Berkeley, April 2001.
2. A. Rowstron, P. Druschel, *Pastry: Scalable, distributed object location and routing for large-scale peer-to-peer systems*, IFIP/ACM International Conference on Distributed Systems Platforms (Middleware), Heidelberg, Germany, pages 329-350, November, 2001
3. C. Tang, Z. Xu, S. Dwarkadas, *Peer-to-Peer Information Retrieval Using Self-Organizing Semantic Overlay Networks*, ACM SIGCOMM 2003, Karlsruhe, Germany, August 2003.
4. C. Tang, S. Dwarkadas, Z. Xu, *On Scaling Latent Semantic Indexing for Large Peer-to-Peer Systems*, Proc. 27th Annual International ACM SIGIR Conference, Sheffield, UK, July, 2004.
5. E. Cohen, A. Fiat, H. Kaplan, *A case for associative peer to peer overlays*, ACM SIGCOMM Computer Communication Review, Volume 33, Issue 1 (January 2003).
6. I. Stoica, R. Morris, D. Karger, M.F. Kaashoek, H. Balakrishnan, *Chord: A scalable peer-to-peer lookup service for internet applications*, In Proc. ACM SIGCOMM 2001, August 2001.
7. D. Karger, E. Lehman, F.T. Leighton, M. Levine, D. Lewin, R. Panigrahy. *Consistent hashing and random trees: Distributed caching protocols for relieving hot spots on the World Wide Web.* In Proc. 29th Annual ACM Symposium on Theory of Computing, pages 654-663, May 1997.
8. H. Cai, J. Wang, *Peer-to-peer computing: Foreseer: a novel, locality-aware peer-to-peer system architecture for keyword searches*, Proc. the 5th ACM/IFIP/USENIX international conference on Middleware, Oct 2004.
9. J. Li, B. T. Loo, J. M. Hellerstein, M. F. Kaashoek, D. Karger, R. Morris, *On the feasibility of peer-to-peer web indexing and search.* In 2nd International Workshop on Peer-to-Peer Systems (IPTPS), 2003.
10. J. Lu, J. Callan. *Content-based retrieval in hybrid peer-to-peer networks*, Proc. the 12th international conference on Information and knowledge management (CIKM), Pages: 199 - 206.
11. M. Aneiros, V. Estivill-Castro, C. Sun, *Social browsing: Group unified histories an instrument for productive unconstrained co-browsing*, Proc. 2003 International ACM SIGGROUP Conference on Supporting Group Work, Nov 2003.
12. M.W. Berry, Z. Drmac, E.R. Jessup, *Matrices, Vector Spaces, and Information Retrieval*, SIAM Review, pages 335-362, June 1999.

13. M. Li, W.C. Lee, A. Sivasubramaniam, D.L. Lee, *A Small World Overlay Network for Semantic Based Search in P2P*, 2nd Workshop on Semantics in Peer-to-Peer and Grid Computing.
14. M. Zaki, S. Parthasarathy, M. Ogihara, W. Li. *New algorithms for fast discovery of association rules.* Proc. the 3rd Int'l Conf. Knowledge Discovery and Data Mining (KDD), 1997.
15. O. Gnawali, *A keyword-set search system for peer-to-peer networks.* Master's thesis, Massachusetts Institute of Technology, 2002.
16. Onestat.com, *Most People Use 2 Word Phrases in Search Engines According to OneStat.com*, available at http://www.onestat.com/html/aboutus_pressbox27.html
17. P. Reynolds, A. Vahdat, *Efficient peer-to-peer keyword searching.* In Proceedings of ACM/IFIP/USENIX Middleware Conference, volume 2672, pages 21-40, Rio de Janeiro, Brazil, June 2003.
18. P. Francis, T. Kambayashi, S. Sato, S. Shimizu, *Ingrid: A Self-Configuring Information Navigation Infrastructure*, 4th International World Wide Web Conference, December 11-14, 1995.
19. R. Agrawal, R. Srikant, *Fast algorithms for mining association rules*, Proc. 20th International Conference on Very Large Data Bases (VLDB), pages 487-499, Morgan Kaufmann, 1994.
20. S. Ratnasamy, P. Francis, M. Handley, R. Karp, S. Shenker, *A scalable content-addressable network*, In Proc. ACM SIGCOMM 2001, August 2001.
21. *TREC relevance judgments*, http://trec.nist.gov/data/reljudge_eng.html.
22. Y. Shao, R.Y. Wang, *BuddyNet: History-Based P2P Search*, 23-37, ECIR, 2005.
23. Z. Wu, W. Meng, C.T. Yu, Z. Li, *Towards a Highly-scalable and Effective Metasearch Engine*, Proc. 10th International World Wide Web Conference, 2001.